Design by Contract

Contracts

- Alice (client) hires Bob (server) to fix her car.
- They make a contract.
  - Alice agrees to give Bob $100 in advance
  - Bob agrees that when he is done, the car will be in good working order

Contracts

	Obligation	Benefit
Alice (client)	Must pay $100	Has working car
Bob (server)	Must fix car	Has $100 to buy materials

Contracts in programming

- Consider a method **tan** to compute the tangent of an angle between 0 and 89 degrees.
- Alice will write the client.
- Bob will implement **tan**.
- Contract
  - Syntactic signature: **double** tan(**double** x)
  - Alice agrees to send, as argument, a value between 0 and 89.
  - Bob agrees that the result will be equal to the tangent to at least three decimal places.

Contracts in programming

	Obligation	Benefit
Alice (client)	Must supply an argument in range [0,89]	Result equals the tan of the argument to 3 decimal places
Bob (implementer)	Must ensure the result equals the tan of the argument to 3 decimal places	Argument in range [0,89]

Contracts in programming

From the point of view of the implementer:

- the clients obligation is what is **required**.
- the implementer's obligation is what the implementation **ensures**.
- We document procedures as follows

```
/**
* requires boolean expression
* ensures boolean expression
*/
```

"requires clause" -- also called "precondition"

"ensures clause" – also called "postcondition"

Contracts in programming

We use “result” to represent the result of an invocation.

- For example:

```
class DegMath {
    /** requires 0 <= x &&  x <= 89
    * ensures result == tan(x)  to three decimal places,
    * where tan is the mathematical tangent function in
    * terms of degrees. */
    static double tan( double x )
}
```

If the client breaks the contract

- Given this contract, the client can not make any assumptions about what a call such as

  DegMath.tan(90.0)   or   DegMath.tan(-1.0)

  might do.
- *The client is obligated to ensure that the expression in the requires clause is true at the **start** of the invocation.*

If the client respects the contract

- For the implementation to be correct, the implementer must ensure that

  ```
  DegMath.tan( 25 )
  ```

  equals the mathematically correct value to 3 decimal places.
- *The implementer is obligated to ensure that the "postcondition" is true, but only in those cases where the "precondition" is true.*

Final values

- The requires clause refers to the values of expressions at the time start of the invocation.
- In the ensures clause, we must often refer to both the initial values of expressions and the final values of expressions.
- We use the convention that

  expression'

  means the value of the expression at the end of the invocation. Often the expression is a variable.

Final values

- Example

```
class Point {
    double x, y ;

    /** requires true
    * modifies x, y
    * ensures x'==0.0 && y'==0.0
    Point() { … }

    /** requires true
    * modifies x, y
    * ensures x' == x + deltaX && y' == y + deltaY
    */
    void move( double deltaX, double deltaY ) {…}
    …
}
```

Omitting the requires clause

- In this example, there is no obligation on the caller beyond the syntactic signature. We can omit the requires clauses

```
class Point {
    double x, y ;

    /** modifies x, y
    * ensures x'==0.0 && y'==0.0
    Point() { … }

    /** modifies x, y
    * ensures x' == x + deltaX && y' == y + deltaY
    */
    void move( double deltaX, double deltaY ) {…}
    …
}
```

Framing

- The **modifies** clause is used to indicate which variables may be changed by a method.

```
class Point {
    double x, y ;

    /** modifies x
    * ensures x' == x + deltaX
    */
    void moveLeft( double deltaX ) {…}
    …
}
```

Example

- Partitioning a list

```
/** requires a != null and a.length > 0
* modifies a[*]
* ensures a[*]' is a permutation of a[*] and
* 0 <= result and result < a.length and
* (for all i in {0,1,…,a.length-1}
*        (i < result implies a[i]' <= a[result]') and
*        (i > result implies a[i]' >= a[result]') )
**/
int partition( double[] a )
```

Further reading

- The paper that introduced the term "design by contract" was

  Meyer, Bertrand. "Applying 'design by contract'." *Computer* 25, no. 10 (1992): 40-51.

- The ideas date back to the late 60s and 70s. For example, the Euclid programming language, designed in 1977, had support for pre- and postconditions.
- Meyer's paper was important for applying the ideas to object-oriented programming.